docproc Documentation

Release 2018

Marcus von Appen

Mar 25, 2018

Contents:

docproc is a simple content processing pipeline, which allows you to take arbitrary input data and to transform it to create output data of any kind.

docproc consists of a set of applications, which allow you to perform different transformation steps one after each other to achieve the desired result. Its design is based on the functional steps to be taken to get useful output out of raw data and can be described as follows:

1. consume input
2. process output based on technical and functional requirements for the desired output
3. output the processed content as necessary, by e.g. sending it to a different host, writing it to disk or consuming the result via a queue.

To enable scalability, each of those functional steps can be handled by a separate application of docproc. The applications are connected by message queues, they read from and write to. This allows you to scale individual parts or complete processing pipelines as required by your input and output scenarios.

CHAPTER 1

Quick Start

The following steps will run a small docproc content processing pipeline on your local machine and transform simple CSV input into HTML files.

We are using NATS as message queue implementation for your local system. Follow its installation instructions to get it installed.

1. Start gnatsd in a separate shell:

```
$ gnatsd
```

2. Create two new directories for in- and output named `data` and `output`:

```
$ mkdir data
$ mkdir output
```

3. Start docproc.fileinput in a separate shell using the NATS example configuration. docproc.fileinput will watch the directory named `data` in the current directory:

```
$ docproc.fileinput -c examples/docproc-fileinput-natsio.conf
```

4. Start docproc.proc in a separate shell using the NATS example configuration. docproc.proc will write processed contents into a directory named `output` in the current directory:

```
$ docproc.proc -c examples/docproc-proc-natsio.conf
```

5. Copy the the test records CSV file into the data directory to start content processing:

```
$ cp examples/data/testrecords.csv data/testrecords.csv
```

6. To verify that everything worked as expected, check the `data` directory for the now processed `testrecords.csv.DONE` file and the `output` directory for a set of new HTML files.

For a more sophisticated setup, take a look at the *Docker Setup* section.

CHAPTER 2

Installing

This section provides an overview and guidance for installing docproc.

2.1 Binary Releases

You can download pre-built binary releases of docproc for different platforms from https://github.com/marcusva/docproc/releases. If your platform is not listed, you can also build docproc from source.

2.2 Building From Source

You can download source snapshots of docproc from https://github.com/marcusva/docproc/tags. Besides the source distribution, you also will need the following tools:

- Golang 1.8 or newer (https://golang.org/)
- dep (https://golang.github.io/dep/)

docproc relies on a message queue implementation. It currently supports the following message queues:

- beanstalk - http://kr.github.io/beanstalkd/
- NSQ - http://nsq.io/
- NATS - https://nats.io/

Unpack the source snapshot into your *GOPATH*, run the *dep* command and build docproc.

On Unix and Linux run

```
$ tar xzvf docproc-.tar.gz $GOPATH
$ cd $GOPATH/github.com/marcusva/docproc
$ dep ensure
$ build-release.sh
```

On Windows run

```
> unzip docproc-.zip %GOPATH%
> cd %GOPATH%/github.com/marcusva/docproc
> dep ensure
> build-release.bat
```

Those commands will build a set o docproc release distributions in the *dist* folder.

2.3 Installation

Unpack the matching distribution package for your operating system and copy the required binaries into the desired target location.

Example for Windows:

```
> unzip docproc-0.0.1-windows-amd64.zip
> cd docproc-0.0.1-windows-amd64
> copy docproc*.exe C:\docproc\bin
```

Example for Linux:

```
$ unzip docproc-0.0.1-linux-amd64.zip
$ cd docproc-0.0.1-linux-amd64
$ cp docproc*. /usr/local/bin
```

Set up the configuration files as appropriate and you are good to go.

CHAPTER 3

Applications

docproc consists of multiple tools, which cover different aspects and steps of the content processing chain. The very first step, consuming input data from other applications, is covered by *docproc.fileinput*. Subsequent steps, such as transforming the input into the desired target format, are all handled by *docproc.proc*.

3.1 docproc.fileinput

docproc supports processing content from files, such as CSV or SAP RDI, via the *docproc.fileinput* application.

When invoking docproc.fileinput, you may specify any of the following options:

```
docproc.fileinput [-hv] [-c file] [-l file]
```

-c <file>
: Load the configuration from the passed file. If not provided, docproc.fileinput will try to read from the configuration file `docproc-fileinput.conf` in the current working directory.

-h
: Print a short description of all command-line options and exit.

-l <file>
: Log information to the passed file.

-v
: Print version details and information about the build configuration and exit.

3.1.1 Configuration

The configuration file uses an INI-style layout and contains several sections, some of them being optional and some of them being mandatory.

```
[log]
# log to a specific file instead of stdout
# file=<path/to/the/file>
# level can be one of Emergency,Alert,Critical,Error,Warning,Notice,Info,Debug
level = Info

# Queue to write the read messages to
[out-queue]
type = nsq
host = 127.0.0.1:4150
topic = input

# Enabled file input handlers
[input]
handlers = rdi-in, csv-in

# SAP RDI file handler
[rdi-in]
format = rdi
transformer = RDITransfomer
folder.in = data
pattern = *.gz
interval = 2

# CSV file handler
[csv-in]
format = csv
transformer = CSVTransformer
delim = ;
folder.in = data
pattern = *.csv
interval = 2
```

Logging

Logging is configured via the `[log]` section. It can contain two entries. The `[log]` section is optional and, if omitted, logging will happen on STDERR with the log level `Error`.

file The file to use for logging. This can be a file or writable socket. If omitted, STDERR will be used.

level The log level to use. The log level uses the severity values of RFC-5424 in either numerical (`0`, `3`, ...) or textual form (`Error`, `Info`, ...). If omitted, `Error` will be used.

```
[log]
file = /var/log/docproc-fileinput.log
level = Info
```

Note: It is recommended to use the log level `Error` in a production environment to spot message processing issues (e.g. a queue being not reachable anymore). In rare situations, docproc.fileinput may use a more severe log level to indicate critical internal application problems.

Output Queue

The output queue to write messages, generated from the input files, to, is configured via the `[out-queue]` section. Configuration entries for the queue may vary slightly, depending on the used message queue provider. The following entries are required nevertheless.

```
[out-queue]
type = nsq
host = 127.0.0.1:4150
topic = input
```

type The message queue type to use. This can be one of

- `beanstalk`
- `nats`
- `nsq`

host The host or URI to use for connecting to the queue. The exact connection string to use varies, depending on the queue type and your service layout.

topic The message queue topic to write to. Consumers, such as docproc.proc can use the same topic to receive and process the incoming messages from docproc.fileinput.

File Input

File input handlers are activated in the `[input]` section and configured in an own, user-defined section. The `[input]` section tells docproc.fileinput, which other sections it shall read to configure the appropriate handlers.

The currently supported handlers are explained in *File Input Handlers*.

handlers A comma-separated list of sections to use for configuring and activating input handlers. The entries must match a section within the configuration file.

```
[input]
# Set up two handlers, which are configured in [rdi-in] and [csv-in]
handlers = rdi-in, csv-in

[rdi-in]
...

[csv-in]
...
```

3.2 docproc.proc

Processing content is mainly done by docproc.proc, with some minor exception for docproc.fileinput, which feeds file-based data into the processing queues being handled by docproc.proc.

When invoking docproc.proc, you may specify any of the following options:

```
docproc.proc [-hv] [-c file] [-l file]
```

-c <file>
Load the configuration from the passed file. If not provided, docproc.proc will try to read from the configuration file `docproc-proc.conf` in the current working directory.

-h
Print a short description of all command-line options and exit.

-l `<file>`
Log information to the passed file.

-v
Print version details and information about the build configuration and exit.

3.2.1 Configuration

The configuration file uses an INI-style layout.

Logging

Logging is configured via the `[log]` section. It can contain two entries.

file The file to use for logging. This can be a file or writable socket. If omitted, STDERR will be used.

level The log level to use. The log level uses the severity values of RFC-5424 in either numerical (`0`, `3`, ...) or textual form (`Error`, `Info`, ...). If omitted, `Error` will be used.

```
[log]
file = /var/log/docproc-fileinput.log
level = Info
```

The `[log]` section is optional and, if omitted, logging will happen on STDERR with the log level `Error`.

Note: It is recommended to use the log level `Error` in a production environment to spot message processing issues (e.g. a queue being not reachable anymore). In rare situations, docproc.fileinput may use a more severe log level to indicate critical internal application problems.

In-, Output and Error Queue

The input queue to read messages from is configured via the `[in-queue]` section. The output queue to write processed messages for other consumers is configured via the `[out-queue]` section. If you want to preserve messages, that failed to process, you can also coonfigure an error queue via the `[error-queue]` section.

Configuration entries for the queue(s) may vary slightly, depending on the used message queue provider. The following entries are required nevertheless.

```
[in-queue]
type = nsq
host = 127.0.0.1:4161
topic = input

[out-queue]
type = nsq
host = 127.0.0.1:4150
topic = output

[error-queue]
type = nsq
```

```
host = 127.0.0.1:4150
topic = error
```

type The message queue type to use. This can be one of

- `beanstalk`
- `nats`
- `nsq`

host The host or URI to use for connecting to the queue. The exact connection string to use varies, depending on the queue type and your service layout.

topic `[in-queue]`: The message queue topic to read messages from for processing.

`[out-queue]`: The message queue topic to write to. Consumers, such as following docproc.proc instances can use the same topic to receive and process the incoming messages.

`[error-queue]`: The message queue topic to write failed messages to.

Processors

Processors are activated in the `[execute]` section and configured in an own, user-defined section. The `[execute]` section tells docproc.proc, which other sections it shall read to configure the appropriate handlers.

handlers A comma-separated list of sections to use for configuring and activating processors. The entries must match a section within the configuration file. The processors are executed in the order of appearance.

Processing the message stops immediately, if one of the configured processors cannot sucessfully process the message. If an `[error-queue]` is configured, docproc.proc will write the message in its current state to that queue.

```
# Processors
[execute]
handlers = add-data, xml-transform

# Processor of type "ValueEnricher"
[add-data]
type = ValueEnricher
rules = /app/rules/preproc/testrules.json

# Processor of type TemplateTransformer"
[xml-transform]
type = TemplateTransformer
output = _xml_
templates = /app/templates/preproc/*.tpl
templateroot = main
```

The currently supported processors are explained in the chapter *Processors*.

CHAPTER 4

File Input Handlers

docproc.fileinput comes with support for converting different file formats and file content into processable messages, which can be individually activated and configured.

It currently supports the conversion of the following input formats:

- SAP RDI spool files via the *RDITransformer*
- CSV data via the *CSVTransformer*

Each individual file handler shares a common set of configuration entries:

```
[your-config]
format = <format-name>
folder.in = <directory to check>
pattern = <file pattern to check>
interval = <check interval>
transformer = <the relevant input transformer>
# additional, transformer-specific configuration entries
```

format The input file format. This is mainly used for informational purposes within the message's metadata and does not have any effect on the message processing.

folder.in The directory to watch for RDI files to be processed.

pattern The file pattern to use for identifying RDI files. This can be a wildcard pattern, strict file name matching or regular expression that identifies those files, that shall be picked up by the `RDITransformer`.

interval The time interval in seconds to use for checking for new files. This must be a positive integer.

transformer The input transformer to use. See below for a list of currently available input transformers.

4.1 Input Transformers

RDITransformer Processes SAP RDI spool files and transforms the contained documents into messages.

CSVTransformer Processes CSV files and transforms the contained rows into messages.

4.1.1 RDITransformer

SAP RDI files can be read using the `RDITransformer` within the input handler configuration:

```
[your-config]
format = rdi
...
transformer = RDITransfomer
```

RDI files picked up by the `RDITransfomer` are assumed to be Gzip-compressed, regardless of their suffix.

When processing an RDI file, the RDITransformer creates one or more messages. A new message will be created, whenever a header ('H') window is found. All data windows following the header will be put into the same message until the next header window is found or if there are no more data windows to read.

```
+-----------------------------------------------------------+--------------+
| RDI contents                                              | docproc      |
|                                                           | Message      |
+===========================================================+==============+
| H0123456789789789...                                      |   Message 1  |
| S                                                         |              |
| CCODEPAGE ...                                             |              |
| C...                                                      |              |
| CCODEPAGE 1100                                            |              |
| DMAIN      SECTION_A                ABCD ...              |   Content    |
| DMAIN      SECTION_A                FIELDX ...            |     of       |
| DMAIN      SECTION_A                FIELDQ ...            |  Message 1   |
| ...                                                       |              |
|                                                           +==============+
| H0123456789789789 ...                                     | Message 2    |
| S                                                         |              |
| CCODEPAGE ...                                             |              |
| C...                                                      |              |
| CCODEPAGE 1100                                            |              |
| DMAIN      SECTION_A                ABCD ...              |   Content    |
| DMAIN      SECTION_A                FIELDX ...            |     of       |
| DMAIN      SECTION_A                FIELDQ ...            |  Message 2   |
| C...                                                      |              |
| DMAIN      SECTION_B                FIELD99 ...           |              |
| ...                                                       |              |
+-----------------------------------------------------------+--------------+
```

Control ('C') and sort ('S') windows will be skipped and have no effect on the message order or content layout.

Note: The `RDITransformer` follows an all-or-nothing approach when processing an RDI file. The created messages are only placed on the queue, if the whole RDI file can be read and transformed sucessfully.

The resulting message(s) consist of a content section, which contain one or more `sections` entries named after the data window that contains the fields.

The above example would produce the following messages.

Message 1

```
{
    "metadata": {
        "format": "rdi",
        "batch": 1517607828,
```

```
        "created": "2018-02-02T22:43:48.0220047+01:00"
    },
    "content": {
        "sections": [
            {
                "name": "SECTION_A",
                "content": {
                    "ABCD": "...",
                    "FIELDX": "...",
                    "FIELDQ": "...",
                }
            }
        ]
    }
}
```

Message 2

```
{
    "metadata": {
        "format": "rdi",
        "batch": 1517607828,
        "created": "2018-02-02T22:43:48.0220047+01:00"
    },
    "content": {
        "sections": [
            {
                "name": "SECTION_A",
                "content": {
                    "ABCD": "...",
                    "FIELDX": "...",
                    "FIELDQ": "...",
                }
            },
            {
                "name": "SECTION_B",
                "content": {
                    "FIELD_99": "...",
                }
            }
        ]
    }
}
```

4.1.2 CSVTransformer

CSV files can be read using the `CSVTransformer` within the input handler configuration.

```
[your-config]
format = csv
...
transformer = CSVTransformer
delim = ;
```

delim The column separator to use.

When processing a CSV file, the CSVTransformer creates one or more messages, depending on the amount of rows within the CSV file. The first row is considered the header row and its column values are used as field names for the message content.

The CSV contents

```
CUSTNO;FIRSTNAME;LASTNAME
100112;John;Doe
194228;Manuela;Mustermann
```

would result in two messages to be created:

Message 1

```
{
    "metadata": {
        "format": "csv",
        "batch": 1517607828,
        "created": "2018-02-02T22:43:48.0220047+01:00"
    },
    "content": {
        "CUSTNO": "100112",
        "FIRSTNAME": "John",
        "LASTNAME": "Doe"
    }
}
```

Message 2

```
{
    "metadata": {
        "format": "csv",
        "batch": 1517607828,
        "created": "2018-02-02T22:43:48.0220047+01:00"
    },
    "content": {
        "CUSTNO": "194228",
        "FIRSTNAME": "Manuela",
        "LASTNAME": "Mustermann"
    }
}
```

CHAPTER 5

Processors

docproc's core command, *docproc.proc* offers a set of different, simple processing tools, which can enhance, change, transform or send message contents.

The following pages provide in-depth information about the different processors and their usage.

ContentValidator Validates the message contents against a predefined set of rules.

ValueEnricher Enables docproc to add new content to a message or to modify existing content of the message.

TemplateTransformer Provides templating support via Go's `text/template` package.

HTMLRenderer Provides templating support via Go's `html/template` package. It is similar to the *TemplateTransformer*, except that `html/template` contains some builtin safety nets for HTML content.

FileWriter Writes a specific entry of the message content to a file on disk.

HTTPSender Sends a specific entry of the message content via HTTP POST to an HTTP host.

5.1 ContentValidator

The ContentValidator checks, if the contents of a message conform to a set of predefined rules. This allows docproc to process only those messages, which are considered functionally valid.

5.1.1 Configuration

The ContentValidator requires the following configuration entries:

```
[contentvalidator-config]
type  = ContentValidator
rules = /path/to/a/rules/set
```

type To configure a ContentValidator, use `ContentValidator` as `type`.

rules `rules` refers to a file on disk containing the rules to be executed on the message content. A rule consists of one or more conditions.

5.1.2 Defining Rules

The rules to be executed are kept in a simple JSON-based list.

```
[
    {
        "name": "First rule",
        "path": "NET",
        "op": "less than",
        "value": 0,
    },
    {
        "name": "Second rule",
        "path": "ZIP",
        "op": "exists",
    }
]
```

See *Operating the Rules Engine* for more details about how to configure rules.

5.2 ValueEnricher

The ValueEnricher enables docproc to add new content to a message or to modify existing content of the message. It uses a simple rules engine to conditionally modify or add content.

5.2.1 Configuration

The ValueEnricher requires the following configuration entries:

```
[valueenricher-config]
type  = ValueEnricher
rules = /path/to/a/rules/set
```

type To configure a ValueEnricher, use `ValueEnricher` as `type`.

rules `rules` refers to a file on disk containing the rules to be executed on the message content. A rule consists of one or more conditions and a target value to set.

5.2.2 Defining Rules

The rules to be executed are kept in a simple JSON-based list.

```
[
    {
        "name": "First rule",
        "path": "NET",
        "op": "less than",
        "value": 0,
        "targetpath": "DOCTYPE",
        "targetvalue": "CREDIT_NOTE"
```

```
    },
    {
        "name": "Second rule",
        "path": "ZIP",
        "op": "exists",
        "targetpath": "HAS_ZIP",
        "targetvalue": true
    }
]
```

A rule to be used for the ValueEnricher consists of the following fields:

- `name`, `path`, `op`, `value`
- `targetpath` - *specific to the ValueEnricher*
- `targetvalue` - *specific to the ValueEnricher*

See *Operating the Rules Engine* for more details about how to configure rules. Rules being used by the ValueEnricher contain two additional fields:

targetpath Defines the path to use for writing the provided targetvalue. If the given path does not exist, it will be created. Similarily to the `path`, the targetpath can be nested using a dotted notation.

Accessing arrays is currently not possible.

targetvalue The value to write into targetpath. The value can contain portions of the existing message's content using a `${<sourcepath>}` notation.

5.2.3 Defining Target Paths

Target paths to write content to can be defined in the same way as the source paths for comparision. A target path can refer to an existing path, causing it to be overwritten with the new value on evaluating the rule successfully. The target path can also be a completely new path, that will be created, if the rule is successful.

Let's add a city name based on the provided shortcut for the following message.

Message:

```
{
    "content": {
        "city_sc": "NY"
    }
}
```

Rule:

```
{
    "path": "city_sc",
    "op": "equals",
    "value": "NY",
    "targetpath": "city",
    "targetvalue": "New York"
}
```

Resulting Message:

```
{
    "content": {
```

```
        "city_sc": "NY",
        "city": "New York"
    }
}
```

Overwrite the city's shortcut with the city name

Message:

```
{
    "content": {
        "city": "NY"
    }
}
```

Rule:

```
{
    "path": "city",
    "op": "equals",
    "value": "NY",
    "targetpath": "city",
    "targetvalue": "New York"
}
```

Resulting Message:

```
{
    "content": {
        "city": "New York"
    }
}
```

Add an address block containing the city name.

Message:

```
{
    "content": {
        "city_sc": "NY"
    }
}
```

Rule:

```
{
    "path": "city_sc",
    "op": "equals",
    "value": "NY",
    "targetpath": "address.city",
    "targetvalue": "New York"
}
```

Resulting Message:

```
{
    "content": {
        "city_sc": "NY",
```

```
        "address": {
            "city": "New York"
        }
    }
}
```

5.2.4 Defining Target Values

Target value can be any kind of atomic value types, such as integers, decimal numbers, boolean values or strings. More complex values, such as JSON objects, maps or arrays are not supported.

Message:

```
{
    "content": {
        "CITY": "New York",
        "ZIP": "10006",
    }
}
```

Rule:

```
{
    "path": "ZIP",
    "op": "exists",
    "targetpath": "HAS_ZIP",
    "targetvalue": true
}
```

```
{
    "content": {
        "CITY": "New York",
        "ZIP": "NY-10006",
        "HAS_ZIP": true
    }
}
```

Furthermore, target values can copy the values from existing paths, as long as those contain atomic value types. To refer to an existing path, use `${}`.

Prefix the ZIP code with state information for New York:

Message:

```
{
    "content": {
        "CITY": "New York",
        "ZIP": "10006",
    }
}
```

Rule:

```
{
    "path": "CITY",
    "op": "equals",
    "value": "New York",
```

```
    "targetpath": "ZIP",
    "targetvalue": "NY-${ZIP}"
}
```

Resulting Message:

```
{
    "content": {
        "CITY": "New York",
        "ZIP": "NY-10006"
    }
}
```

5.3 TemplateTransformer

The TemplateTransformer provides templating support via Go's text/template package. This allows docproc to create complex, message-dependent content to be stored in the message itself.

5.3.1 Configuration

The TemplateTransformer requires the following configuration entries:

```
[templatetransformer-config]
type = TemplateTransformer
identifier = path_to_store
templates = /path/to/all/templates/*.tpl
templateroot = main
```

type To configure a TemplateTransformer, use `TemplateTransformer` as `type`.

identifier The path to use on the message's content to store the transformed result in.

templates Location of the template files on disk. This should be a glob pattern match.

templateroot The template entry point to use (`{{define "your-templateroot" }}`).

5.4 HTMLRenderer

The HTMLRenderer provides templating support via Go's html/template package. This allows docproc to create complex, message-dependent content to be stored in the message itself. It is similar to the *TemplateTransformer*, except that `html/template` contains some builtin safety nets for HTML content.

5.4.1 Configuration

The HTMLRenderer requires the following configuration entries:

```
[htmlrenderer-config]
type = HTMLRenderer
identifier = path_to_store
templates = /path/to/all/templates/*.tpl
templateroot = main
```

type To configure a HTMLRenderer, use `HTMLRenderer` as `type`.

identifier The path to use on the message's content to store the transformed result in.

templates Location of the template files on disk. This should be a glob pattern match.

templateroot The template entry point to use (`{{define "your-entrypoint" }}`).

5.5 FileWriter

The FileWriter writes a specific entry of the message content to a file on disk.

5.5.1 Configuration

The FileWriter requires the following configuration entries:

```
[filewrite-config]
type = FileWriter
identifier = htmlresult
rules = /app/rules/output/file-html.json
filename = filename
path = /app/output
```

type To configure a FileWriter, use `FileWriter` as `type`.

identifier The path of the message's content save to the file.

filename The path of the message's content containing the filename to use.

path The directory to use for storing the file.

rules The set of rules to use to decide, if the file shall be written or not.

5.6 HTTPSender

The HTTPSender sends a specific entry of the message content via HTTP POST to an HTTP host.

5.6.1 Configuration

The HTTPSender requires the following configuration entries:

```
[httpsender-config]
type = HTTPSender
identifier = body
url = http://some.endpoint/receive_msg
```

type To configure a HTTPSender, use `HTTPSender` as `type`.

identifier The path of the message's content to send to the host.

url The URL to send the content to.

CHAPTER 6

Operating the Rules Engine

docproc ships with a small, easy-to-use rules engine that allows many of its builtin processors to behave in certain ways, according to your message contents. The rules are, if not stated otherwise executed against the message's `content` section.

Since docproc uses JSON heavily, rules are also expressed in a JSON notation and are organised as a simple JSON array:

```
[
    {
        ... # rule 1
    },
    {
        ... # rule 2
    },
    {
        ... # rule 3
    }
]
```

6.1 Rule Configuration

A rule typically consists of the following fields.

```
{
    "name":     "<optional name of the rule>",
    "path":     "<message content path to use for comparing or checking>",
    "op":       "<operator to use>",
    "value":    "<value to use for comparision>",
    "subrules": [ "<more nested rules>" ]
}
```

name An optional name describing the rule. This is for maintenance purposes and does not have any effect on the rule, if provided or absent.

path The message's content element to check. Paths can be nested using a dotted notation.

op The comparision operator to use. If not stated otherwise, the comparision will consider path being the left-hand and value the right-hand argument:

```
value-of-path <op> rule-value
```

value The value to compare the path's value against. **value** can be omitted, if the comparision operator is `exists` or `not exists`. If it is provided for those operators, it will be ignored.

subrules A list of additional rules that have have to be tested. The rule as well as all its sub-rules have to match successfully to consider the rule as a whole as successful.

Please note that all **subrules** are evaluated before the rule itself is evaluated. Thus, the most inner subrule is the first being tested.

6.2 Setting Paths

Paths are always relative to the message's content element and can use a dotted notation to access nested elements of a message. It is also possible to access array values using brackets and the required index number.

Let's have a look at a few examples of configuring proper paths for rules. Given the following message

```
{
    "content": {
        "name": "John Doe",
        "age": 30,
        "address": {
            "street": "Example Lane 123",
            "zip": "10026",
            "city": "New York"
        },
        "netValues": [
            1000.00,
            453.00,
            -102.00,
            2
        ]
    }
}
```

you can access and check the age of John Doe being greater than 20 via

```
{
    "path": "age",
    "op": ">",
    "value": 20
}
```

Accessing nested elements is done by connecting the element and its sub-element with a dot. To check, if an address exists and if its city is New York, you can use `address.city`.

```
{
    "path": "address.city",
    "op": "eq",
    "value": "New York",
    "subrules": [
```

```
        {
            "path": "address",
            "op": "exists"
        }
    ]

}
```

Note: Subrules are evaluated before the rule itself is evaluated. Thus, if you think of multiple conditions that have to apply, you have to build them in a reverse order:

```
1st condition:      if an address exists
2nd condition:      and if its city name is "New York"
```

thus becomes:

```
2nd (outer) rule:   and if its city name is "New York"
1st (inner) rule:   if an address exists
```

Make use of `name` to explain more complex rules to keep your maintenance efforts at a minimum.

You can access array values using brackets `[]` and the value's index. Indexing starts at zero, so that the first element can be accessed by `[0]`, the second by `[1]` and so on.

```
{
    "path": "netValues[2]",
    "op": ">=",
    "value": 500,
}
```

6.3 Operators

Existence To check, if a given path of a message exists (it may contain nil values or empty strings) or not, use the `exists` and `not exists` operators:

```
{
    "path": "address",
    "op": "exists"
}

{
    "path": "alternativeName",
    "op": "not exists"
}
```

Any value provided on the rule, will be ignored.

Equality The following operators check, if the provided values are equal:

`=`, `==`, `eq`, `equals`

```
{
    "path": "name",
    "op": "=",
```

```
    "value": "John Doe",
}
```

Their counterparts, to check for inequality, are:

`<>`, `!=`, `neq`, `not equals`

```
{
    "path": "name",
    "op": "neq",
    "value": "Jane Janeson",
}
```

Size Comparators Values can also be compared by size. This is straightforward for numeric values. If you use size comparators on strings, please note that the strings are compared lexicographically.

To check, if the left-hand value is greater than the right-hand value:

`>`, `gt`, `greater than`

```
{
    "path": "age",
    "op": ">",
    "value": 21,
}
```

To check, if the left-hand value is greater than *or equal to* the right-hand value:

`>=`, `gte`, `greater than or equals`

```
{
    "path": "netValues[0]",
    "op": "gte",
    "value": 500,
}
```

Their counterparts for checking the other way around:

`<`, `lt`, `less than`

```
{
    "path": "age",
    "op": "<",
    "value": 50,
}
```

and

`<=`, `lte`, `less than or equals`

```
{
    "path": "netValues[3]",
    "op": "less than or equals",
    "value": -1.0,
}
```

String Matching To check, if a string contains another string or not, use the following operators:

`contains`, `not contains`

```
{
    "path": "name",
    "op": "contains",
    "value": "Doe",
}
{
    "path": "name",
    "op": "not contains",
    "value": "Jane",
}
```

As for the size comparators, this checks, if the left-hand value contains the right-hand value. To check the other way around, use

`in`, `not in`

instead.

```
{
    "path": "name",
    "op": "in",
    "value": "John Doe, Jane Doe, or their kids",
}

{
    "path": "address.city",
    "op": "not in",
    "value": "London, Vancouver, Washington, Halifax",
}
```

CHAPTER 7

Docker Setup

The following information will guide you through a simple configuration scenario for creating your own docker setup. They can be summed up as

1. create the base image via docker
2. build the docproc images via docker-compose
3. run everything via docker-compose

Note: You will need the source distribution of docproc, which you can find at https://github.com/marcusva/docproc/tags for stable snapshots.

7.1 Base Image

The docproc base image contains all docproc applications as well as a `nsqd` binary to get docproc up and running for testing with the NSQ message queue system.

Create the base image with the following instructions:

```
$ docker build -t docproc/base .
```

The base image is now registered in your local docker registry as `docproc/base`.

Note: The docproc applications of the base image will be built with nsq support only. To change this behaviour, you can tweak the `BUILD_FLAGS` within `Dockerfile` as necessary or override the `BUILD_FLAGS` at the command line:

```
$ docker build --build-arg BUILD_FLAGS="-tags beanstalk" -t docproc/base .
```

The nsqd binary will be built and installed nevertheless, if `Dockerfile` is not edited, though.

7.2 Build docproc Images

Create all docproc images with the following instruction:

```
$ docker-compose build
```

This creates the following set of docproc images:

- `docproc/fileinput`
- `docproc/preproc`
- `docproc/renderer`
- `docproc/postproc`
- `docproc/output`

Each image can be run individually. Each image runs a local `nsqd` server to be used by the individual docproc executable.

7.3 Run Everything

All services, including an nsqd, nsqlookupd and nsqadmin instance can be run via:

```
$ docker-compose up
```

Todo: document ports and directories properly.

Index

Symbols

C